# Exercises: ASP.NET Core

# Assign Roles to Users,Implementing Areas

Problems for exercises for the ["ASP.NET Core Advanced" course @ SoftUni](https://softuni.bg/trainings/4954/asp-net-advanced-june-2025)

A popcorn and film reels and a movie ticket

Description automatically generated with medium confidence

## IdentityUser – Movie Relation

In order to implement the Watchlist feature, we need a way to **connect users to the movies** they have added to their Watchlist.

### Creating the Data Model

This is a classic **many-to-many relationship**:

* **One user** can add **many movies** to their Watchlist
* **One movie** can appear in the Watchlists of **many users**

To model this relationship in our database, we will create a new table called **UserMovie**, which will serve as a **mapping table** between IdentityUser and Movie.

*At this stage, we are still using the default* IdentityUser *class for user identity management. Therefore, our UserMovie model will reference* IdentityUser*. In a future workshop, when we extend Identity, this model will be easily adaptable to work with* ApplicationUser*.*

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

### Adding the UserMovie DbSet in DbContext

This ensures that Entity Framework Core knows about the UserMovie entity and can create the corresponding table.

A screen shot of a computer program

AI-generated content may be incorrect.

### Create a Configuration File

Next, we will configure the **composite primary key** and the relationships by using a clean   
**Fluent API configuration file**.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

### Understanding the Fluent API

In this project, **we are using the Fluent API to configure the relationships between our entities**.

* The **Fluent API allows us to use C# code to describe how Entity Framework Core should map our entity classes** to database tables.
* This is an **alternative to using only data annotations** (like **[Key]**, **[ForeignKey]**, **[Required] ... etc.**) inside the entity classes.

### Elements of the Fluent API in Our UserMovieConfiguration

* HasKey() — Composite Primary Key  
  The primary key of the UserMovie table is made up of two columns: UserId and MovieId.
* **HasOne() → WithMany() → HasForeignKey()** — Relationship to User and Movie
* OnDelete(DeleteBehavior.Cascade)
  + If a **user is deleted** → **all related Watchlist entries will also be deleted** automatically
  + If a **movie is deleted** → **all related Watchlist entries will be deleted** automatically

### Apply Migration and Update the Database

Now that we have fully configured the UserMovie entity, it is time to apply this change to the database.

In the previous subsection, we wrote a clean Fluent API configuration in the UserMovieConfiguration class. We also applied this configuration in the OnModelCreating method of the CinemaAppDbContext. The next step is to **generate a new EF Core migration** so that Entity Framework can **update our actual database schema**.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

When this command completes, **Visual Studio will generate a new migration class under the Migrations folder** in your project. The migration will contain the instructions needed to create the UsersMovies table.

The next step is to **update the database** with this migration.

A screenshot of a computer program

AI-generated content may be incorrect.

After this command finishes successfully, the database will be updated. You can verify this by opening **SQL Server Management Studio** (SSMS) or the database explorer inside Visual Studio.

A screenshot of a computer

AI-generated content may be incorrect.

## Implementing Roles

Roles define the **permissions or responsibilities** assigned to a user in an application (e.g., Admin, Manager, User). Roles are used to **restrict or allow access** to specific parts of an application.

### Configure Identity for role Support

* Add Identity Role Support: Ensure the AddIdentity call in Program.cs includes IdentityRole<Guid>  
  A screen shot of a computer program

  Description automatically generated
  + IdentityRole<Guid> ensures roles are stored in the database
  + AddRoles<IdentityRole<Guid>> registers role management services

### Seed Roles in the Database

Create a **dedicated seeder class** in the Configuration folder  
A screenshot of a computer

Description automatically generated

A computer screen shot of text

Description automatically generated



**Update Program.cs**: Call the seeding method after building the app:

A screen shot of a computer code

Description automatically generated

**Update the routing configuration** to include areas:

A computer code with text

Description automatically generated with medium confidence

* Place the areas route **before** the default route so the application checks for areas first.

### Verify the Seeded Roles Database Provider (SSMS)

After running the application with the SeedRoles method, we need to confirm that the roles have been successfully added to the database. This ensures that the role-seeding logic worked correctly and that the roles (Admin, Manager, User) are available for use in the application.

The roles are stored in the AspNetRoles table, which is part of the default Identity schema created by ASP.NET Core. Checking this table in SQL Server Management Studio (SSMS) helps verify the success of the operation.

A screenshot of a computer

Description automatically generated

### Assign Roles to a User

Now that roles (Admin, Manager, User) **have been seeded into the database**, the next step is to **assign a role to a specific user**. For example, **we can assign the Admin role to a user with the email admin@example.com**. This ensures that the user can access functionality restricted to the Admin role

* **Creating the user** (if they do not already exist)
* **Assigning the desired role to the user**

**Update the DatabaseSeeder Class:**

A computer screen shot of a website

Description automatically generated



Call the AssignAdminRole Method in Program.cs

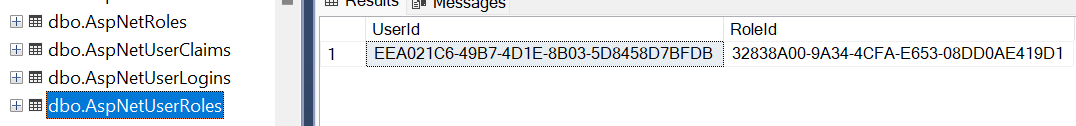
A screenshot of a computer program

Description automatically generated

Restart the Application to execute the updated DatabaseSeeder logic:

A screenshot of a computer

Description automatically generated



## Add Role-Based Redirects

### Update the Middleware in Program.cs

In the Program.cs file, **add middleware to check the user’s role after they authenticate and redirect them**.

* Place this logic **after** app.UseAuthentication() and **before** app.UseAuthorization()

A computer code with text

Description automatically generated

## Implementing Areas

### Allow Admin to Assign Roles to Users

The Admin will be able **manage user roles** directly from the **Admin area**:

* Listing all users in the system
* Selecting a specific user to assign or update roles
* Saving the role assignment in the database

### Create the Admin Area

The **Admin Area** will serve as a dedicated section of the application where **only users with the Admin role can access and perform administrative tasks**, such as assigning roles to users and managing key aspects of the system.

* Inside the existing **Areas folder**, add a **new folder** named **Admin**

A screenshot of a computer

Description automatically generated

### Setup the Admin Area Structure

**Inside the Admin folder, add the following subfolders**

* **Areas/Admin/Controller**
* **Areas/Admin/Views**
  + **/Index**
  + **/UserManagement**

**A screenshot of a computer

Description automatically generated**

### Add \_ViewStart.cshtml in the Admin Area

**Add a new file** named \_ViewStart.cshtml in the **Areas/Admin** folder

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

This ensures that **all views inside the Admin area inherit from the shared layout**

### Add \_ViewImports.cshtml in the Admin Area

A screenshot of a computer

Description automatically generated



### Add the Admin Controllers

* Inside **Areas/Admin/Controllers**, **create a new controller** named HomeController
* Update the controller code to **specify it belongs to the Admin area**

A screenshot of a computer program

Description automatically generated

* Inside **Areas/Admin/Controllers**, **create a new controller** named UserManagementController

**A screenshot of a computer

Description automatically generated**

### HomeController

The HomeController is the entry point for the Admin Area and should provide an **overview** and **navigation to other sections** like UserManagement, CinemaManagement, and MovieManagement.

### Create First View for the Admin Area

**Admin dashboard View:**

* Inside **Areas/Admin/Views/Home**, create Index.cshtml

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* **Add the following content**

****

**A screenshot of a computer

AI-generated content may be incorrect.**

### Plan the User Management Features

* View a list of users
* Assign or revoke roles for each user
* Delete users if necessary

### Update the UserManagementController

A screenshot of a computer program

AI-generated content may be incorrect.

**Purpose of Index Action**: It retrieves **a list of users from the database** using the UserManager service and   
passes it to the view.

### Create the View Model for User Management

* **Create a new foder** named ViewModels in Areas/Admin folder
* Inside the ViewModels folder, **add a new class** named UserViewModel.cs

A screenshot of a computer program

Description automatically generated

### Update the UserManagementController to Use the View Model

* Update the Index action to use the UserViewModel for displaying user data

A screenshot of a computer program

Description automatically generated

### Create the Initial View for User Management

* Navigate to **Areas/Admin/Views/UserManagement**
* Inside the **UserManagement** folder, create a new file named Index.cshtml

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated



A screenshot of a computer

AI-generated content may be incorrect.

### Cinema Management

* Manage Cinemas



A screen shot of a computer

AI-generated content may be incorrect.

* Add a New Cinema (Create)



A screenshot of a computer

AI-generated content may be incorrect.

* Edit a Cinema



A screen shot of a computer screen

AI-generated content may be incorrect.

* Delete a Cinema

A screen shot of a computer

AI-generated content may be incorrect.

The admin can toggle a cinema's status between active and deleted. This is a soft delete, meaning the cinema is not removed from the database but simply marked as deleted

A screenshot of a computer

AI-generated content may be incorrect.

### Movie Management

* Manage Movies

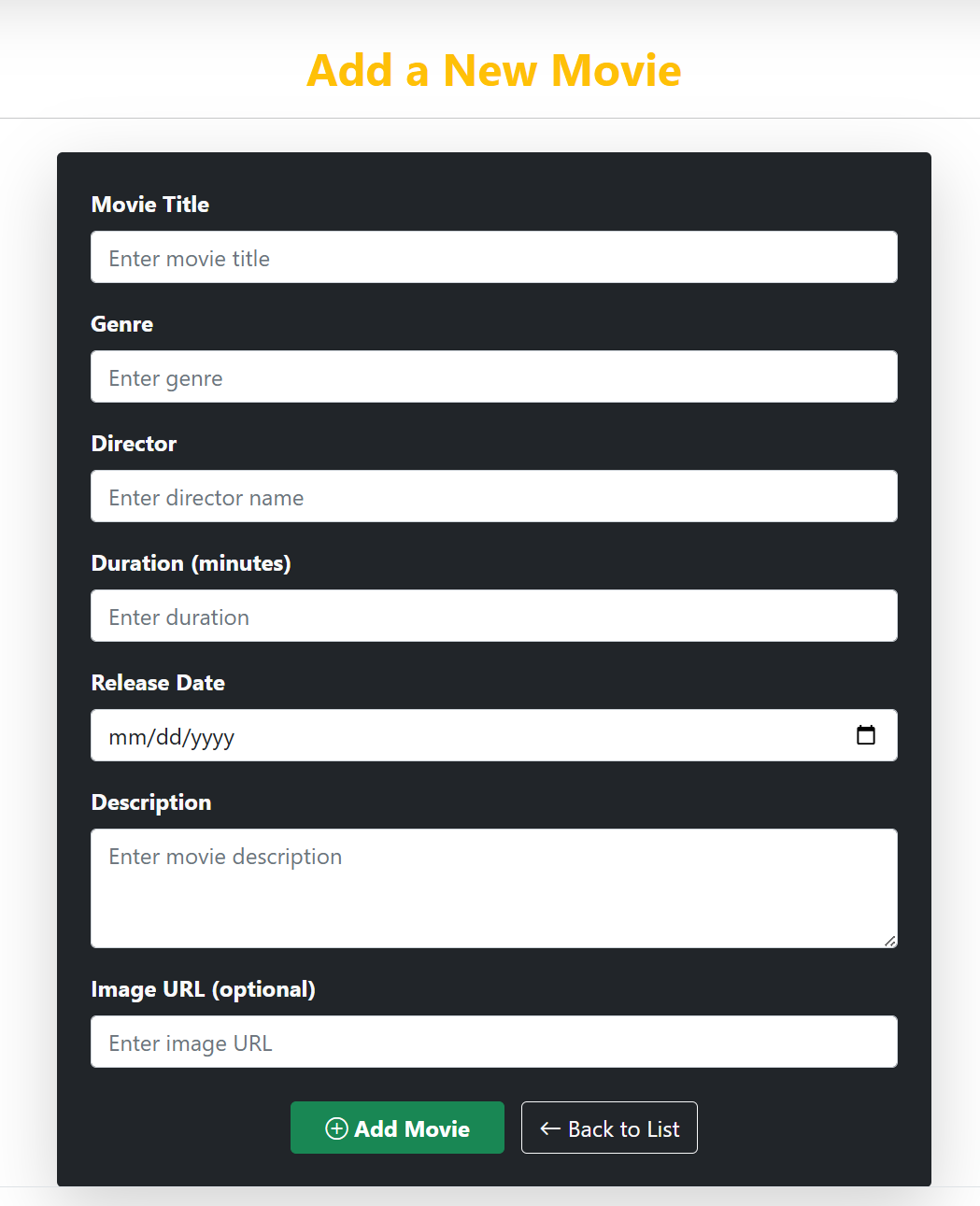


A screenshot of a computer

AI-generated content may be incorrect.

* Add New Movie





* Edit a Movie



A screenshot of a movie

AI-generated content may be incorrect.

* Delete a Movie

### Update the \_Layout.cshtml for Admin Navigation



Add a navigation dropdown for Admin users to access the Admin Dashboard and its sections.

A screenshot of a computer code

AI-generated content may be incorrect.

When an authenticated user with the "Admin" role logs in, they will see an “Admin” dropdown in the navigation bar with quick access to:

* **Manage Users**
* **Manage Cinemas**
* **Manage Movies**

A screenshot of a computer

AI-generated content may be incorrect.